

Multi-view pattern matching

Matthias Gallé

July 19, 2016

Abstract

We introduce the *multi-view pattern matching* problem, where a text can have multiple views. Each view is a string of the same size and drawn from disjoint alphabets. The pattern is drawn from the union of all alphabets.

The algorithm we present is an extension of the Horspool algorithm, and in our experiments on synthetic data it shows an $3\times$ improvement over the naive baseline.

1 Motivation

Suppose we are given a text, together with its Parts-of-Speech (POS) annotated sequence, and want to retrieve all occurrences of *Carlson invented NN*. Another example is the retrieval of records for a collection of trivia for instance: supposing an additional second layer of annotation where each word is annotated if it is a superlative (*SUP*) or not, and we are interested in the occurrences of *NNS is the SUP NNS of France* (Mt-Blanc, highest, mountain; Bettencourt, wealthiest, person, Saint-Cirq-Lapopie, prettiest, village; etc).

Although pattern matching is one of the most studied problems in computer science, currently there are no methods to solve this problem directly. Regular expression in particular are not suited for this kind of problem because the variety lies in the sequence, not in the pattern.

We call this the *multi-view pattern matching* problem, because we consider that there are k different views over the sequence S , and the pattern can contain any combination of these views. In this paper we present an efficient method that address the problem of finding all occurrences of p in such a multi-view sequence S by pre-processing p .

1.1 Problem Definition

We are given k sequences $S = [s_1, \dots, s_k]$, all of size n and over pair-wise disjoint alphabets ($s_i \in \Sigma_i^*$; $\Sigma_i \cap \Sigma_j = \emptyset, \forall i \neq j$). We denote by $t(c)$ the type of character c : $t(c) = k$ iff $c \in \Sigma_k$.

A pattern $p \in \left(\bigcup_{i=1}^k \Sigma_i \right)^*$ occurs at position i of S if $p[j] = s_{t(p[j])}[i + j]$, $\forall 1 \leq j \leq |p|$. The problem we then want to solve is to find all occurrences of p in S .

2 Related Work

In broad terms, pattern matching algorithms divide into two classes: if the sequence S is considered stable and several different patterns will be searched on it it may be more efficient to index the sequence S . But if the pattern p is fixed and it will be queried over different sequences, or p is significantly shorter than S it is convenient to pre-process the pattern instead. Here we focus on this second case.

There are several different exact string matching algorithms, and many variations on each of them¹. The two most famous ones are probably the Knuth-Morris-Pratt [4] and the Boyer-Moore [1] ones. Both improve over a brute-force algorithm by taking advantage of the information of a mismatch, shifting over parts of S on which one can be certain that a match will not occur.

To see the intuition behind, consider such a brute-force algorithm, that tries to match p at each position i of S . If for instance $p = abc$ and $S = abdabc$, the first alignment would fail because S has a d in its third position, compared to an c for p . The pattern would then be shifted by one, and bda (substring starting at position 2 of S) would be compared to p . However, at this stage we already know that S does not contain any a (the first letter of the pattern) in its three first position, so aligning p to any of these positions fail for sure. KMP is based upon this idea, creating a *failure table* at pre-processing time that gives for each position j of p the position to look for a new match if the current one failed. So, when $ababc$ is matched against $abababc$, the first alignment (at position 1 of S) will fail at position 5 of p ($c \neq a$), but the failure table will indicate that the next alignment should start at position 3 of S , because it observed there ab which coincides with the start of p .

As can be seen from that example, this algorithm takes advantage of repetitions inside p (in particular repetitions of any prefix of p). This becomes even more explicit in the case of the Boyer-Moore algorithm. The original one had two different tables, and the second one focuses precisely on handling such repetitions of suffixes (because BM compares from right to left). However such repetitive patterns are unusual in practice. This is precisely the insight of Horspol [3] who shows how a simple version of BM (dropping the second rule, and further simplifying the first one) provides sometimes results even better than BM because it shortens the pre-processing time. Fig. 1 extracted from [2] compares different exact simple string pattern matching (Boyer-Moore-Horspool there correspond to [3]).

¹these lecture notes for instance describe 28 such algorithms: <http://www-igm.univ-mlv.fr/~lecroq/string/index.html>

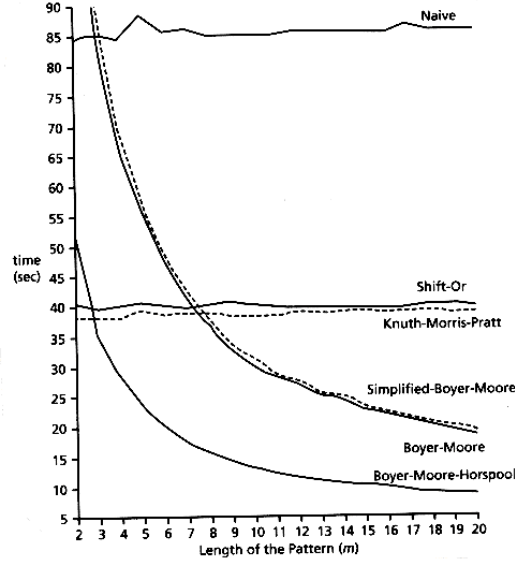


Figure 1: Comparison of exact string matching algorithms (extracted from [2]). This uses random generated text, but the figure for English text in the book is almost equivalent.

Much work has been done to support inexact (or fuzzy) pattern matching, including cases when p includes wildcards (fixed or extensible), multi-character symbols and regular expressions. However, for the precise case where the variations are in S under the form we are considering here we were not able to find any work addressing it.

3 Multi-View Horspool

We decided to focus on the Horspool algorithm and extending it for the multi-view case for two main reasons:

1. Its good performance on benchmarks (see again Fig. 1)
2. The improvement of other algorithms with respect to it lie in the fact that they avoid comparisons due to repetitions inside the pattern. However in our case of multi-view patterns, this is not so straightforward as the following examples shows.

$p = AaAab$ and $S =$

1	2	3	4	5	6	...
b	a	a	a	a	b	...
A	A	A	A	B	B	...

Aligning p at position 1 fails due to the last character. However, if we would take advantage of the repetition of Aa and shift p directly to the position 3, we would be losing the match that occurs at position 2. The fact that symbols from one alphabet may mask symbols from another makes such an approach much more cumbersome. They act as a kind of wildcard in the pre-processing (but not during the matching), and existing extensions of pattern matching to such cases are very sensitive to the number of such wildcards (in our case, any character acts as wildcard for the other alphabets)².

Like in Horspool's algorithm, the pre-processing we propose consists in filling a table β (for *bad character*, the name of the first rule of BM) that gives for each symbol the offset from the end of its latest occurrence. For the previous example this would be $\beta(a) = 1, \beta(A) = 2, \beta(b) = 5, \beta(B) = 1$ (the last occurrence of A is at position $4 = 6 - 2$). Note that for the characters occurring in the last position it gives the offset to the second-to-last occurrence (the reason for this will become clear later on), or $m = |p|$ if it does not occur. In general

$$\beta(c) = \min\{i \mid i = m \vee (1 \leq i < m \wedge p_{t(c)}[m - i] = c)\} \quad (1)$$

The algorithm we propose is then as follows (see also Alg. 1). For each alignment, we first compare the last character and, if it coincides, we then continue comparing the remaining ones. This is equivalent to the classical Horspool algorithm. The novelty lies in the shift, where we consider the lowest value of all the offsets of symbols occurring at the current position, taken over all sequences s_k .

Algorithm 1

mv-horspool(S, p, t)

Input: sequence $S = [s_1, \dots, s_k]$, pattern p , and implicit type function t

Output: all matches of p in S

```

1: pre-process  $p$  to produce  $\beta$  holding Eq. 1
2:  $j := 0$ 
3:  $n, m := |S|, |p|$ 
4: while  $j \leq n - m$  do
5:    $c := S_{t(p[m])}[j + m]$ 
6:   if  $p[m] = c \wedge p[1 : m - 1]$  matches  $S[j : j + m - 1]$  then
7:     output  $j$ 
8:   end if
9:    $j := j + \min_k \beta(S_k[j + m])$ 
10: end while
```

Consider its execution on the following example:

$p = BAbB$ (and therefore $\beta(B) = 3, \beta(A) = 2, \beta(b) = 1$ and $m = 4$ for all other symbols); and $S =$

²see for instance the problems in these lecture notes from Jeff Erickson's class at Urbana-Champaign <http://www.cs.uiuc.edu/class/fa06/cs473/lectures/18-kmp.pdf>

1	2	3	4	5	6	7	8
<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>C</i>	<i>B</i>

In the first alignment, the last character already does not match:

1	2	3	4	5	6	7	8
<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>C</i>	<i>B</i>
<i>B</i>	<i>A</i>	<i>b</i>	<i>B</i>				

For the shift we consider the lowest value of $\beta(b)$ and $\beta(A)$ which is 1 and start the second alignment:

1	2	3	4	5	6	7	8
<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>C</i>	<i>B</i>
	<i>B</i>	<i>A</i>	<i>b</i>	<i>B</i>			

The last character matches, but not the remaining one (here we suppose that we compare from the beginning to the end). p is then shifted according to $\min(\beta(s_1[5], s_2[5]))$, which is 3 (as $\beta(a) = m = 4$). This finally results in a match:

1	2	3	4	5	6	7	8
<i>c</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>C</i>	<i>B</i>
		<i>B</i>	<i>A</i>	<i>b</i>	<i>B</i>		

4 Results

Because Alg. 1 has to compare the values at the current position of all k sequences, it is not clear that this should be faster than the naive algorithm, which is independent of k .

We therefore compare the performance over randomly generated text, using the following setting: $k = 3, n = 100\,000, |\Sigma_i| = 10$ and varying m from 2 to 30. We implemented both algorithms in *C*, and measured user time over 10 000 generated examples (same sequences for each algorithm, generated uniformly and i.i.d). The results can be appreciated in Fig. 2

While the improvement is not so impressive compared to the standard case (with has improvement of up to x8, see Fig. 1), the performance improvement is still of a factor of 3 times better than the naive algorithm.

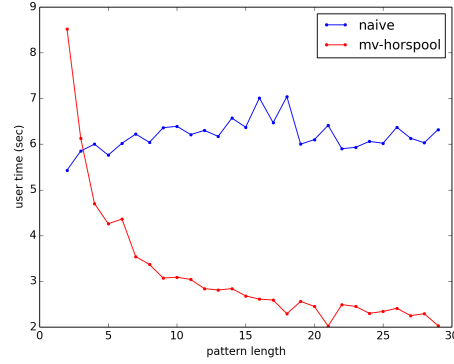


Figure 2: Total user time for 10 000 instances of multi-view pattern matching.

References

- [1] Robert S. Boyer and J Strother Moore. A fast string searching algorithm. *Comm. ACM*, 20(10), October 1977.
- [2] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
- [3] R. N. Horspool. Practical fast searching in strings. *Software - Practice and Experience*, 10:501–06, 1980.
- [4] Donald Knuth, jr Morris, James H., and Vaughan Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.